ScALPEL: A Scalable Adaptive Lightweight Performance Evaluation Library for application performance monitoring

Hari K. Pyla, Bharath Ramesh, Calvin J. Ribbens and Srinidhi Varadarajan
Department of Computer Science, Virginia Tech.
{harip, bramesh, ribbens, srinidhi}@vt.edu

Abstract

As supercomputers continue to grow in scale and capabilities, it is becoming increasingly difficult to isolate processor and system level causes of performance degradation. Over the last several years, a significant number of performance analysis and monitoring tools have been built/proposed. However, these tools suffer from several important shortcomings, particularly in distributed environments. In this paper we present ScALPEL, a Scalable Adaptive Lightweight Performance Evaluation Library for application performance monitoring at the functional level. Our approach provides several distinct advantages. First, ScALPEL is portable across a wide variety of architectures, and its ability to selectively monitor functions presents low run-time overhead, enabling its use for large-scale production applications. Second, it is run-time configurable, enabling both dynamic selection of functions to profile as well as events of interest on a per function basis. Third, our approach is transparent in that it requires no source code modifications. Finally, ScALPEL is implemented as a pluggable unit by reusing existing performance monitoring frameworks such as Perfmon and PAPI and extending them to support both sequential and MPI applications.

1. Introduction

Millions of dollars are spent each year in building faster HPC systems to reduce computation time for a wide range of computational science and engineering applications. However, only a few select benchmarks achieve anything close to peak performance on these high-end resources, with most applications running at a small fraction of the advertised peak speed. Increasing the usable capacity of high-end computational resources necessitates the use of performance measurement and analysis tools that provide detailed sensor data to guide algorithm redesign and optimization.

Performance analysis of even sequential single node applications is complicated by several factors including cache hierarchy, data placement, resource usage and TLB misses to name a few. This problem is significantly exacerbated for large-scale parallel applications that face variations in scheduling and message transfer latency.

Simple UNIX commands and basic tools like *time* and gprof [1] provide preliminary information such as the total amount of time spent in a particular code segment. However, in order to optimize the application this information is not sufficient. The total time reported by these tools is a function of several factors which are often not visible in the source code but happen dynamically when the code is executed on a particular platform. Hardware performance counters provide valuable insights into various performance aspects of applications; they report the "cause(s)" rather than just the "effect(s)".

Traditional tools such as Perfmon [2], PAPI [3], [4], Perfsuite [5] and many others are commonly used to monitor hardware counters. While such tools are quite useful, they suffer from several shortcomings.

First, such tools/libraries monitor only a fixed set of events throughout the lifetime a program. Modern x86 processors only allow monitoring of four events at best. These events are hard-coded while profiling the program. Hence, fully exploring the desired event space[1] requires a time consuming iterative process of compiling and running the application many times. Moreover, this process may also require modifications to the source code in order to accommodate a different set of events. Additionally, some of the tools [2], [3], [4], [6], [7] are not easily usable, since they involve a learning curve to understand the API they provide.

In addressing these issues, some of the tools [2], [5] provide transparency through statistical code sampling and time sharing software multiplexing techniques [8], [9], [10], [2]. While such techniques are useful in exploring the event space, they suffer from tradeoffs involving high accuracy (high sampling rate) and low performance overhead (low sampling rate).

Most importantly, such techniques are not suitable for a wide range of applications because they report the counter values during a particular phase. This is a serious limitation. Consider a scenario where an application is iterative in nature with varying phases during iterations. Multiplexing events in time may not be synchronized with the phase of the application and hence the true nature of the application

1. The set of all counters the performance analyst is interested in monitoring.

is not captured.

Second, existing tools do not facilitate a complete runtime configuration of hardware performance counters. The counters are defined at compile time and cannot be modified during execution of the program.

Third, most of tools are based on techniques that involve a significant overhead. These techniques commonly involve interrupts, timers, break-points, etc. Such techniques affect the critical path of execution and often result in significant monitoring overhead.

Finally, most of the tools do not provide runtime access to the counters. The raw values obtained from the counters are usually reported after program termination. The lack of such information prevents applications from making any runtime decisions based on performance characteristics. Libraries such as PAPI and Perfmon provide an API to read the counters. However, the counters cannot be accessed asynchronously since the API function calls must be embedded in the code and hence must be specified at compile time.

Even though there are a plethora of performance monitoring tools, analyzing the performance of parallel applications is still a tedious task for two reasons. First, parallel applications are inherently complex. Second, as discussed previously, the tools are not flexible enough and do not simplify the task of performance analysis. In order to simplify this problem and address the above mentioned issues, we present a compiler directed tool called ScALPEL to monitor hardware performance counters. *Our intent is not to develop yet another performance monitoring tool, but instead to develop an approach that can be easily combined with existing tools such as Perfmon and PAPI and make them more usable.*

In this paper, we propose a simple solution that addresses the above problems and makes the following contributions:

- **Simple:** *We provide an approach that simplifies the process of performance monitoring.*
- **Lightweight:** *We propose and demonstrate a new compiler driven technique that reduces the performance monitoring overhead significantly, compared to traditional approaches.*
- **Dynamic:** *We provide a technique that facilitates configuration and access of hardware performance counters at runtime.*
- **Portable:** *We propose an approach that is portable and transparent and can be seamlessly plugged into existing libraries.*

The rest of the paper is organized as follows, we discuss related work in Section 2. We present the details of our approach and implementation in Section 3. In Section 4 we present our evaluation, case study and experimental results. In Section 5 we present the limitations of our approach and finally in Section 6 we summarize our conclusions.

2. Related Work

To place our work in the context of existing research and to help clearly understand our contributions, we classify existing approaches based on the level of their implementation (software and hardware).

Software tools can be classified into libraries, tools and simulators. Libraries such as PAPI [3], [4], Perfmon (*libpfm*) [2] and PCL [7] provide APIs that are well documented and easily accessible. However, they suffer from several important drawbacks. First, they do not offer any transparency to the performance analyst. The performance analyst usually has to go through the tedious task of modifying the program and re-compiling it several times to monitor all the counters.

On the other hand, open source tools such as Perfmon (Pfmon) [2], Apple's Shark [6], Perfsuite [5], ProfileMe [11], TAU [12] and commercial tools such as Intel's VTune [13] are usable, transparent and provide elaborate graphical results. However they often suffer from significant profiling performance overhead. Most importantly most of these tools install break points in the application and hence are not suitable for profiling recursive or nested function calls. Our experiments with Perfmon indicate that such techniques can lead to significant performance overheads. Another approach proposed by Zagha et al. [14] provides minimal information at a per process level. Such implementations have lower performance overhead at the expense of granularity and portability. Most importantly, Zagha's implementation is specific to the MIPS architecture and it is not portable across other commonly used architectures such as x86. We take a completely different approach by providing a much more fine-grained, portable and architecture independent solution.

Software tools can also be classified based on the underlying profiling technique. Several approaches [8], [15], [2], [6], [12] are based on statistical sampling. Such implementations are often faced with a choice between accuracy and performance penalty. Our approach does not involve sampling; in fact, we leverage support from the compiler to profile the application.

Other implementations of performance tools rely on simulators [16], [17], [18], [19]. They provide limited information with a limited set of events for the modeled processors. Such an approach is often slow and not scalable across different nodes in a cluster. We choose to implement our technique using real systems running on commonly used architectures; moreover, our approach is easily scalable across multiple cores and nodes in a cluster.

Finally, other approaches [20], [11] include special purpose hardware to monitor the counters. Such techniques incur relatively less performance overhead compared to software based approaches. However, they are not usable in that they require special hardware, which is expensive and usually not a feasible solution to install on all the nodes in a cluster.

Furthermore, they require architecture-specific software to use their hardware. In contrast, our approach is software based and can run as a module on existing tools and libraries.

3. Our approach

In this section we discuss the design and implementation details of ScALPEL. We also explain the rationale behind some of our design choices.

3.1. Overview

We present an overview of our approach in Figure 1a. Our design includes support for instrumenting source code and adaptively configuring hardware performance counters and functions at runtime. We provide a runtime library that implements the instrumentation callbacks and facilitates runtime reconfiguration. We link the application's object code and the runtime library with a pre-existing user-space hardware performance monitoring library to generate the application binary.

Unlike traditional debugging techniques such as breakpoints, interrupts and timers, we use a compile time technique to instrument source code. We choose this approach because of its low overhead and ease of installing callbacks in the object code. From a performance analyst's perspective, while our current implementation requires source recompilation, it does not require any code modifications. The instrumentation occurs at the object code level. We designed our approach to be generic in that it can be extended to other existing compilers and performance monitoring utilities. We choose to use the GNU compiler collection and Perfmon for several reasons. First, they are both open source and are commonly available. While Intel Compilers [21] also support function level instrumentation, they are commercial products and not freely available. Second, the design and interface for the Perfmon library is well written and easy to understand. Finally, extending our approach to include other libraries such as PAPI requires little or no effort.

We present a detailed discussion of ScALPEL's instrumentation methodology and its runtime system in the following subsections.

3.2. Compiler directed instrumentation

The GNU compiler [22] supports function level instrumentation through its code generation options [23]. Based on these options, the compiler instruments either all or selected [2] functions in an application. In the case of selective functional profiling, the performance analyst is required to identify the functions that are of interest and enumerate the rest of the functions as arguments to the compiler command-line. The compiler then instruments these functions with function handlers or callbacks upon function entry and exit. These function handlers are embedded in the object code. For example, as shown in Figure 1b, function *foo* is instrumented by the compiler by inserting callbacks immediately upon entry and just before exit. This concludes the first step where the application is aware of functions that can be monitored.

2. selective functional profiling is available in GCC 4.3.2 onwards

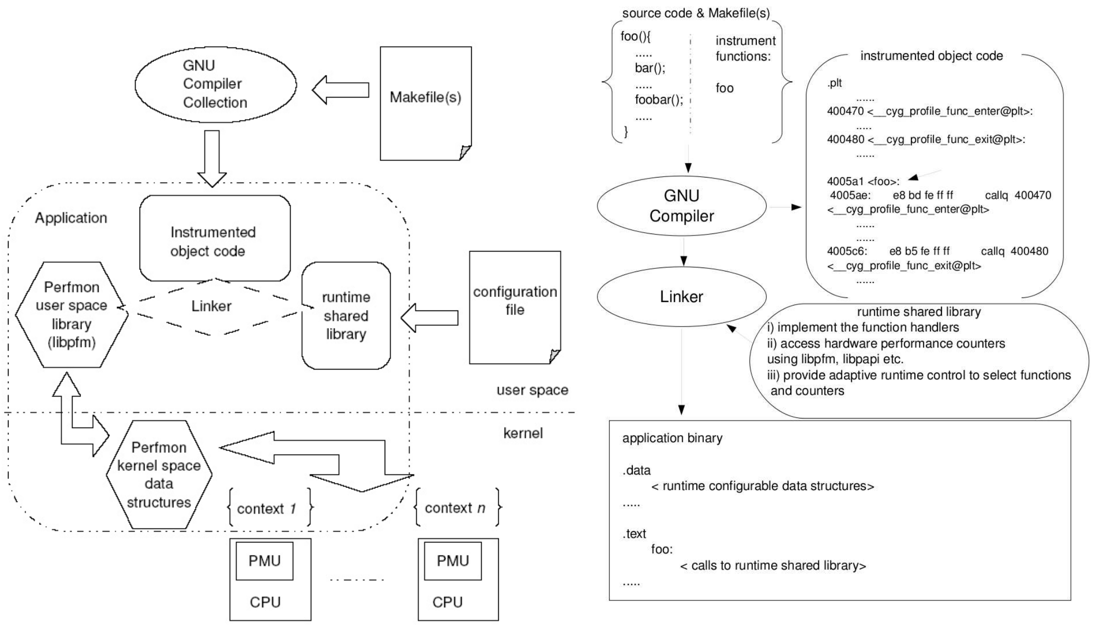
We provide two degrees of freedom to the user: the ability to dynamically change the "events" and the "functions" to monitor at runtime. It is important to note that the list of functions specified at compile time defines the set of all functions that are intercepted. Our design provides enough flexibility to let the user select a subset from this set of functions at runtime. If the set cannot be determined at compile time then an alternative is to choose all functions to be intercepted. This results in additional but minor overhead in most cases. However, the actual overhead is strictly application dependant. We present a detailed discussion of the performance implications of these alternatives in the Section 4.

We define a *context* for each function that the user is interested in monitoring. In ScALPEL a context is centered around a function. The context includes the function name, total number of events, the events, number of subevents for a given event and its corresponding subevents. In Table 1 we present a semantic representation of a typical context. This contextual information is supplied by the user in a configuration file and can be altered at anytime during the application's execution.

3.3. Runtime library

In this section we discuss the next step in performance monitoring, i.e., how ScALPEL actually monitors the hardware counters and how the context information can be changed at runtime. Recall that during compile time, only function handlers are placed in the object code. The ScALPEL's runtime shared library implements these function entry and exit handlers. When it is first loaded the runtime library creates a list of all function contexts described by the user from the initial configuration file.

At runtime, for every function in the set, on entry ScALPEL checks if a context corresponding to this function is specified by the user. If a context does not exist then the function continues executing normally. If a context does exist, then it is loaded and monitoring is initiated for the function. ScALPEL retains the context across any recursive or immediate successive calls to the same function. This helps in reducing the monitoring overhead during recursion and also in situations where a function is called several times repeatedly within a loop. ScALPEL stops monitoring the counters on exit of a function. ScALPEL's runtime library


GNU Compiler Collection
Makefile(s)
Application
Instrumented object code
Perfmon user space library (libpfm)
Linker
runtime shared library
configuration file
user space
kernel
Perfmon kernel space data structures
context 1
context n
PMU
CPU
PMU
CPU
source code & Makefile(s)
foo(){ ..... bar(); ..... foobar(); ..... }
instrument functions: foo
instrumented object code
.plt
400470 <__cyg_profile_func_enter@plt>:
400480 <__cyg_profile_func_exit@plt>:
4005a1 <foo>:
4005ae: e8 bd fe ff ff callq 400470 <__cyg_profile_func_enter@plt>
4005c6: e8 b5 fe ff ff callq 400480 <__cyg_profile_func_exit@plt>
GNU Compiler
Linker
runtime shared library
i) implement the function handlers
ii) access hardware performance counters using libpfm, libpapi etc.
iii) provide adaptive runtime control to select functions and counters
application binary
.data
< runtime configurable data structures>
.text
foo:
< calls to runtime shared library>

(a) overview (b) control flow

Figure 1: Overview and instrumentation process of ScALPEL.

uses the user-space Perfmon library (*libpfm*) to monitor performance counters. The hardware counters are specific to the executing process and not for the entire system. In fact they specifically measure the counters during the scope of a function's execution. The callbacks installed during the instrumentation process identify functions by only their addresses. ScALPEL resolves the addresses with function names by reading the symbol table in the object file to present meaningful results to the user. The results include the name of the function, set of events and their corresponding counter values.

The runtime library is flexible in that it allows the performance analyst to modify the contexts at runtime in several ways. For instance, a new configuration file may be loaded at any time by sending a signal (SIGUSR1) to the application. At this point the runtime system dumps the previous contexts and creates a new set of contexts. New functions can be added or deleted from the list as long as they are from the set specified at compile time. If new functions are added, and whenever they are encountered during execution thereafter, they are monitored. Functions may also be deleted which stops their monitoring. In addition to dynamically configuring the functions and events, ScALPEL also supports multiplexing of events within a function based the number of function calls.

The runtime library contains other data structures that store the counter information and provide easy access to this data at runtime. This allows the performance analyst to make runtime decisions. By default, the values of hardware counters observed during program execution are written to *stdout* upon termination of the program. In our present implementation we choose to use *libpfm* to monitor the hardware counters; however, other libraries such as *libpapi* can be used with minor code modifications to the runtime system.

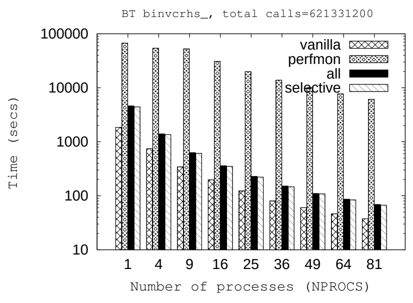
4. Experimental analysis and results

In this section we explain in detail our experimental methodology and the results of our approach. We tested the ScALPEL prototype on the System-G [24] supercomputer running Linux 2.6.27.10 x86_64 kernel with the Perfmon kernel patch [25]. Each node in System G has two quad core Intel Xeon processors running at 2.8 GHz and 8GB of main memory. The nodes are interconnected over a QDR Infiniband switching fabric.

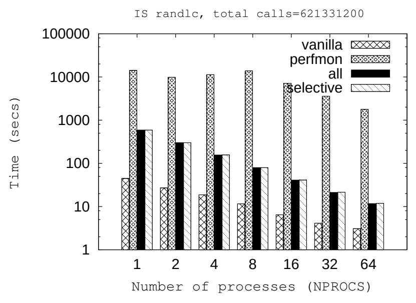
We initially present a performance analysis of our approach compared to Perfmon. We then present a case study to show the flexibility and ease of use of ScALPEL. We used the NAS-Parallel benchmark suite [26] for performance analysis and LINPACK [27] as a case study to help illustrate the adaptivity provided by ScALPEL.

4.1. Performance evaluation

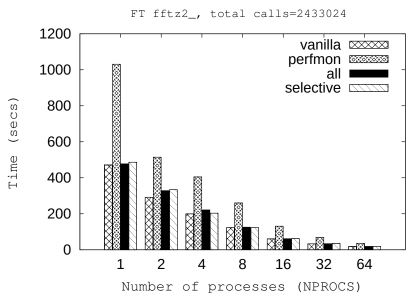
We quantify the performance overhead by measuring the total real time for each execution using the UNIX *time* command. We define the following four test cases;



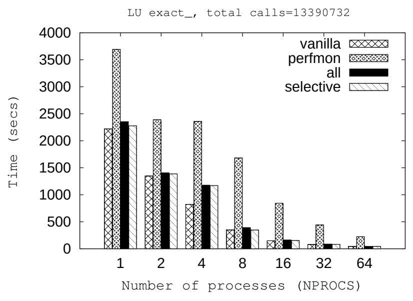
(a) BT (log-lin plot)



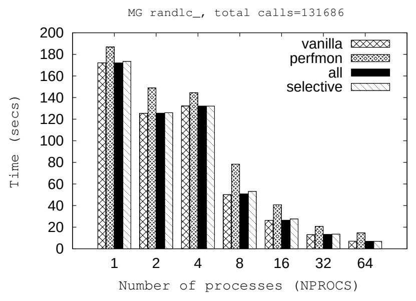
(b) IS (log-lin plot)



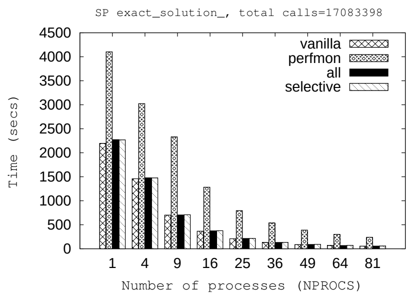
(c) FT



(d) LU



(e) MG



(f) SP

Figure 2: Illustrates the performance overhead of various test cases *vanilla*, *perfmon*, *all* and *selective* along with the selected function and its total number of calls. We choose to use log_{10} scale for BT and IS because the overhead of perfmon was too dramatic and the rest of the cases were negligible in the graph otherwise.

- *vanilla:* does not involve any hardware performance monitoring. The application is run natively
- *perfmon:* measures the hardware counters using Perfmon tool
- *all:* intercepts all the functions in an application, i.e., set of all intercepted but not necessarily monitored functions
- *selective:* intercepts and monitors only a subset of the total functions in an application

In order to fully understand the extent of performance overhead involved in providing complete flexibility to choose both functions and counters, we sub-divide our approach into two categories (*all* and *selective*), depending on the size of the set[3]. The *all* and *selective* test cases describe the worst and best case scenarios respectively of our instrumentation overhead. However, it is important to note that they are not necessarily indicative of the performance counter monitoring overhead. To evaluate the monitoring overhead, we choose to monitor a single function at a time throughout the execution of an application. We choose to do this because it is not possible to obtain the hardware performance counter values for individual functions at a per-function level using *perfmon* without involving sampling. Hence, in the case of *all*, we intercept all functions but monitor only one function, thereby paying the cost of monitoring without any gains from the ability to dynamically change the set of monitored functions. Similarly, in the case of *selective* and *perfmon* we intercept and monitor only one function. We monitored the same set of events and same functions across all four test cases.

We monitored several functions, one at a time per each benchmark. The choice of these functions is based on the number of times they are called during the entire program execution. In Figure 2 we present results for the functions that were called the maximum number of times. Figure 3 illustrates the results for functions which were called on the order of tens of times, hundreds of times and several thousands of times. We choose the NAS CLASS C workload and ran each benchmark with increasing number of processors ($1 <= NPROCS <= 81$). Finally, we ran each benchmark using *perfmon* (pfmon-3.6). We compiled the sources of Perfmon with debugging support disabled (CONFIG_PFMON_DEBUG=n) in the config.mk file and we also commented a *printf* statement that prints "unknown ptrace event" in Line 2045 in *pfmon_task.c*.

We present our findings briefly below:

- In general, *vanilla* took the least amount of time to execute (shown in Figures 2a–2f) compared to the other cases (*perfmon*, *all* and *selective*). This was the

3. Recall that the size of the set indicates the number of functions chosen for profiling at compile time.

Table 1: Layout of a sample configuration file

```
BINARY=my_a.out       // name of the binary
NO_FUNCTIONS=1        // number of functions

[FUNCTION]
FUNC_NAME=foo         // name of the function

NO_EVENTS=2           // total number of events

[EVENT]
ID=DATA_CACHE_MISSES // the event name or id
NO_SUBEVENTS=0        // number of subevents
[/EVENT]

[EVENT]               // begin event information
ID=DISPATCHED_FPU
NO_SUBEVENTS=3
[SUBEVENT]            // list of subevents
ID=OPS_ADD
ID=OPS_ADD_PIPE_LOAD_OPS
ID=OPS_MULTIPLY_PIPE_LOAD_OPS
[/SUBEVENT]
[/EVENT]              // end of event

[/FUNCTION]           // end of function
```

expected result.

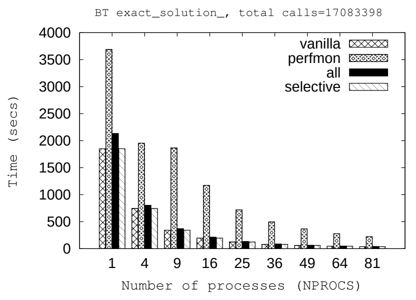
- In scenarios where the number of times a particular function is invoked is relatively small (tens of thousands), the performance monitoring overhead of *perfmon* is comparable to *selective* and *all*, as shown in Figure 2e. This is because the overhead of the underlying technique (breakpoints or compiler instrumentation) is independent of the total life time or scope of a particular function; instead, it depends on the number of times the function is called. Hence, in such scenarios, the actual impact of the hardware performance monitoring methodology is insignificant in the total execution time.
- The overhead of *all* was higher compared to *vanilla* and *selective* for some benchmarks. Since the underlying performance monitoring technique of *all* and *selective* are the same, this overhead is solely attributed to additional cost of intercepting all functions. The extent of this overhead depends on the total number of functions and the number of times each is invoked. We found that in some cases with relatively small number of function calls, (shown in Figures 3g-3h) *all* had slightly more overhead compared to *Perfmon*.
- In general, for a majority of benchmarks the overhead of *perfmon* was much higher than ScALPEL; in some cases (Figures 2a-2b) by two to three orders of magnitude. We found that the *selective* test case had significantly lower overhead compared to *perfmon*. This overhead varies with function call counts in different benchmarks as shown in Figures 2a-2f. The key reason for the reduction in overhead was our use of function interception instead of breakpoints in *perfmon*, which incurs increasing overhead as the monitored function is called repeatedly. Also, the overhead of using *perfmon* to monitor a particular function was higher than the overhead of cumulatively profiling all the functions and monitoring the counters for a given function using *all*. Figure 2 clearly shows that the compiler directed approach provided by ScALPEL provides a low overhead approach to parallel performance measurement.

4.2. Case Study

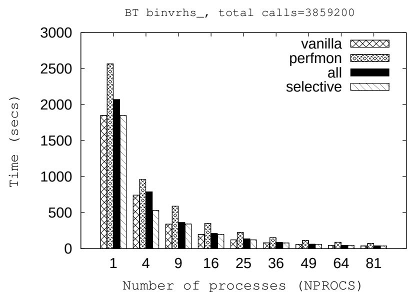
In this section we use a simple case study to illustrate the benefits of runtime reconfigurability as provided by ScALPEL. Two features of ScALPEL are particularly important in this case study, namely the ability to profile only specified functions and the ability to dynamically switch between any number of hardware counters. Existing tools, such as Perfmon, allow users to change the counters of interest at regular specified time intervals. However, switching events at fixed time intervals does not yield accurate performance results for function-level profiling since there may not be any correlation between the time intervals and the function call pattern of the application. In this case study we use ScALPEL to cycle through the event sets of interest after a fixed number of calls to the function(s) of interest. Furthermore, the performance of real applications often depends subtly on interactions among several factors, e.g., instruction level scheduling and parallelism, register pressure, memory hierarchy issues, etc. Hence, it is extremely difficult to know, a priori, which few event counters to track in a given performance gathering run. With ScALPEL, we can gather data from an arbitrary number of performance counters, by cycling through a large set of counters in a predictable way.

As a case study, we use the LINPACK benchmark to compare the performance of two different implementations of the Basic Linear Algebra Subprograms (BLAS), viz. ATLAS [28] and GotoBLAS [29]. The goal here is not to benchmark a system, but rather to study the performance of two implementations of the dominant linear algebra kernel underlying that benchmark code. We do this by analyzing the hardware counters and not the implementations themselves.

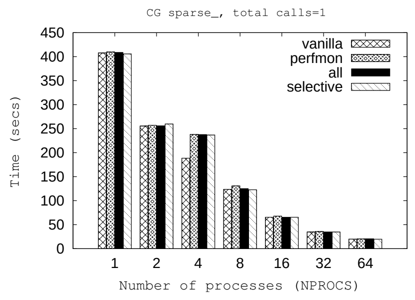
High-performance scientific computations depend to a large degree on the performance of the matrix multiplication kernel. The General Matrix Multiply (GEMM) subroutine in BLAS performs matrix multiplications $C \leftarrow \alpha AB + \beta C$; A, B and C being matrices, while α and β are scalar coefficients. GEMM is often highly tuned to run as fast as possible for high-performance computing as it is the building block for many other routines. GEMM is often used recursively, with input matrices decomposed into smaller



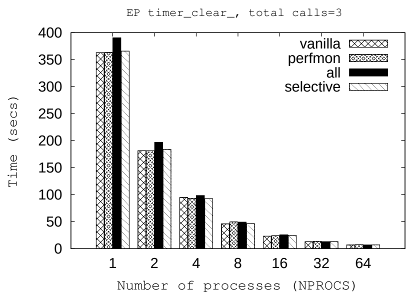
(a) BT



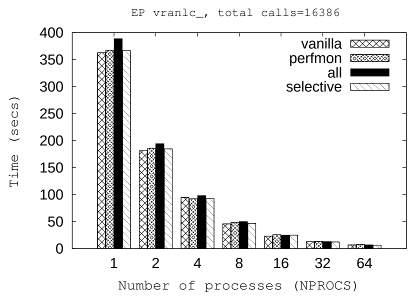
(b) BT



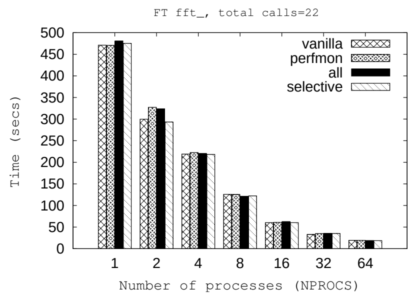
(c) CG



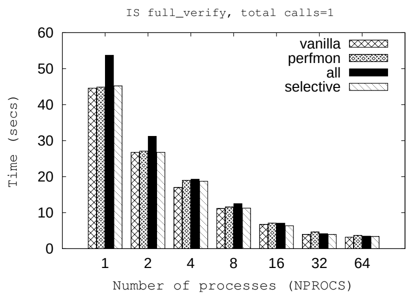
(d) EP



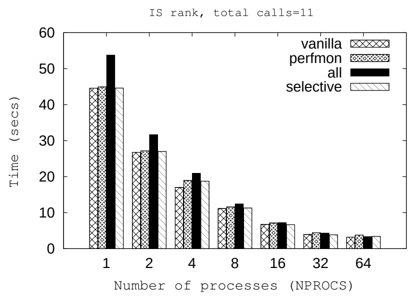
(e) EP



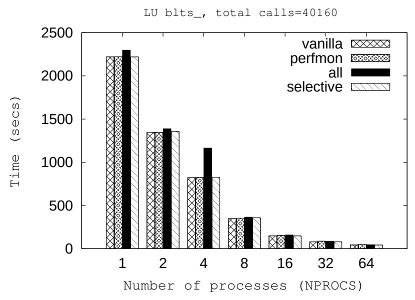
(f) FT



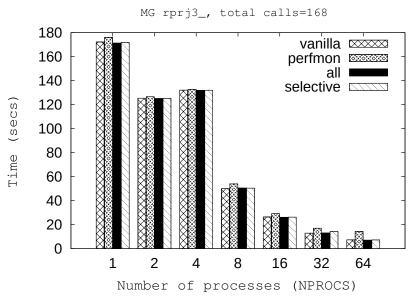
(g) IS



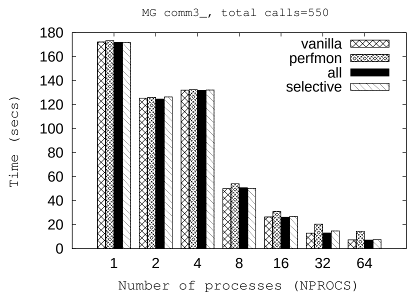
(h) IS



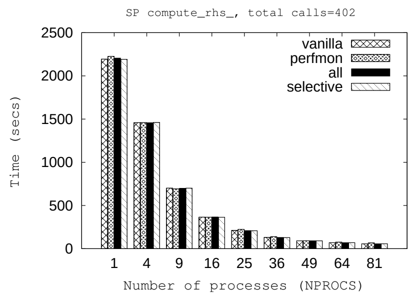
(i) LU



(j) MG



(k) MG



(l) SP

Figure 3: Illustrates the performance overhead of various test cases *vanilla*, *perfmon*, *all* and *selective* along with the selected function and its total number of calls.

block matrices which are in turn operated on using GEMM. Such matrix decompositions allow for better locality of reference, thereby yielding better utilization of system cache. When there is more than one level of cache, blocking can be applied at each level. This technique is one of the optimizations used in the implementation of ATLAS, and based on published reports, one expects the ATLAS implementation of GEMM to perform well in terms of its use of the memory hierarchy. In GotoBLAS, on the other hand, the focus is on minimizing Translation Look-aside Buffer (TLB) table misses. According to [30], "While the importance of cache is also taken into consideration, it is the minimization of such TLB misses that drives the approach." TLB misses are minimized by filling most of the memory addressable by the TLB with the matrix A, while operating on matrices C and B a few columns at a time.

We used ATLAS developmental version 3.9.5 and GotoBLAS version 1.26 in our case study. Both ATLAS and GotoBLAS were compiled to use only a single thread. ATLAS was compiled with both the architectural defaults and full search (*-Si archdef 0*). Five different sets of events were monitored in a single sampling run of the benchmark viz. $\{$*DTLB_MISSES:ANY and L2_LINES_IN:ANY*$\}$, $\{$*L2_RQSTS:ANY and SSE_PRE_MISS:NTA:L1:L2*$\}$, $\{$*L1D_ALL_REF and L1D_ALL_CACHE_REF*$\}$, $\{$*X87_OPS_RETIRED:ANY and SIMD_INST_RETIRED:ANY*$\}$ and $\{$*INST_RETIRED:ANY_P and RESOURCE_STALLS:ANY*$\}$ (all in perfmon2 format). The *ATL_dgemm* function was instrumented in the ATLAS implementation, while the *dgemm_* function was instrumented for GotoBLAS. We cycled through the event sets of interest after every 100 calls to the DGEMM implementation during the sampling run. We also ran the exhaustive case in which we monitored one set of events per run and ran the benchmark five times capturing a different set of events on each run.

Table 2: Hardware counter values for LINPACK run NB = 200, N = 20000 sampling run

Event name	ATLAS (default)	ATLAS (full)	Goto
DTLB_MISSES	2.78e07	2.88e07	4.61e07
L2_LINES_IN	1.65e09	1.56e09	5.72e08
L1D_ALL_REF	2.26e11	2.25e11	1.52e11
L1D_ALL_CACHE_REF	2.26e11	2.25e11	1.52e11
X87_OPS_RETIRED	7.16e05	2.66e05	0.00e00
SIMD_INST_RETIRED	7.13e11	7.13e11	7.11e11
INST_RETIRED	8.19e11	8.19e11	8.76e11
RESOURCE_STALLS	6.39e10	6.35e10	1.57e10

The raw hardware counter values are presented in Table 2 for problem size $N = 20000$ and block size $NB = 200$ with sampling. We do not present the results for the second set of events $\{$*L2_RQSTS:ANY and SSE_PRE_MISS:NTA:L1:L2*$\}$ as the counters returned zeros. In Figure 4 we compare the relative ratio of ATLAS full build and GotoBLAS with the architectural defaults build of ATLAS for both exhaustive and sampling runs. Comparing the exhaustive runs to our function call multiplexed sampling technique shows that the error introduced by sampling is marginal. In terms of bottom-line performance, the benchmark built with GotoBLAS is 9.5% faster than with the default build of ATLAS, while with ATLAS full build we get only an 0.6% improvement over the default build of ATLAS. Normally, all we can do is attribute these performance differences to the cleverness of one implementation over another; but with ScALPEL we can look closer and understand in detail why these observed differences occur, and perhaps identify opportunities for further improvements in this or other codes.

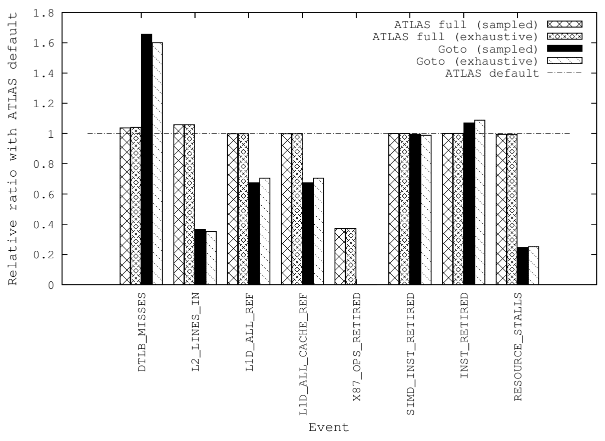


Figure 4: Comparison of ATLAS full build and GotoBLAS.

Recall that the stated goal of the GotoBLAS implementation is to reduce TLB misses. Surprisingly, we see from Figure 4 that GotoBLAS has 65% *more* TLB misses than ATLAS. Looking at other counters, however, we see that GotoBLAS has 65% fewer L2 cache misses than ATLAS, and 75% fewer resource stalls. It seems clear that the GotoBLAS implementation is gaining a performance advantage from these significant reductions in expensive events. Simply counting the total number of TLB misses is misleading. In fact, the Goto implementation incurs substantially more total TLB misses. However, it appears that Goto is able to amortize (or even completely hide) the cost of these misses over a relatively greater amount of useful computation. Said another way, not all TLB misses are created equally. Some may be essentially harmless, and if the TLB misses are managed and scheduled carefully (e.g., with aggressive prefetching), then an implementation may be able to reduce other expensive events (e.g., stalls and cache misses), as happens in this case.

5. Limitations

In this section we discuss some of the limitations of ScALPEL. First, as discussed previously, while our approach does not require any source code modifications, it does require recompilation. Second, we presently support profiling functions that are recursive and have nested calls to other functions. In such situations we monitor the hardware counters for both the function (parent) and its nested call sequence (children). However, we do not support monitoring both the parent and the child at the same time in the same nested call sequence. Third, the granularity of our profiling is restricted to a function level; we do not support profiling at block level. Finally, our present implementation does not report the results on a per thread basis within a process. This is also an issue with Perfmon. In the future we plan to enhance our prototype to isolate the hardware counter values at the granularity of a thread in multi-threaded application.

6. Conclusion

In this paper we addressed several shortcomings of existing performance monitoring tools. We demonstrated that it is possible to monitor the hardware counters using ScALPEL without imposing any significant overhead by using a compiler directed instrumentation technique. Moreover, by coupling the instrumentation technique with our runtime system, we were able to provide an efficient performance monitoring scheme.

We provided a prototype to adaptively configure both functions and events at runtime. We discussed the performance implications of our approach and validated its usage with a case study. We discussed the key issues that lead to performance overhead. Our results indicate that our function instrumentation technique outperforms if not matches other existing tools for a majority of benchmarks. Moreover, for certain benchmarks, both our best and worst case scenarios performed significantly better than Perfmon. And most importantly, we accomplish this without the need for any modifications to an application's source code. Finally, our approach is completely portable and can be used with existing implementations such as Perfmon and PAPI.

References

[1] GNU, "gprof," http://www.gnu.org/software/binutils/.

[2] S. Eranian, "Perfmon2, the hardware-based performance monitoring interface for linux," http://perfmon2.sourceforge.net/.

[3] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci, "A scalable cross-platform infrastructure for application performance tuning using hardware counters," in *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*. Washington, DC, USA: IEEE Computer Society, 2000, p. 42.

[4] J. Dongarra, K. London, S. Moore, P. Mucci, D. Terpstra, H. You, and M. Zhou, "Experiences and lessons learned with a portable interface to hardware performance counters," in *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*. Washington, DC, USA: IEEE Computer Society, 2003, p. 289.2.

[5] R. Kufrin, "Measuring and improving application performance with perfsuite," *Linux J.*, vol. 2005, no. 135, p. 4, 2005.

[6] Apple, "Performance and debugging," http://developer.apple.com/tools/performance.

[7] R. Berrendorf and B. Mohr, "Pcl - the performance counter library:a common interface to access hardware performance counters on microprocessors," http://www.fz-juelich.de/jsc/PCL/doc/pcl/pcl.html.

[8] R. Azimi, M. Stumm, and R. W. Wisniewski, "Online performance analysis by statistical sampling of microprocessor performance counters," in *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*. New York, NY, USA: ACM, 2005, pp. 101–110.

[9] J. M. May, "Mpx: Software for multiplexing hardware performance counters in multithreaded programs," in *IPDPS '01: Proceedings of the 15th International Parallel & Distributed Processing Symposium*. Washington, DC, USA: IEEE Computer Society, 2001, p. 22.

[10] W. Mathur and J. Cook, "Improved estimation for software multiplexing of performance counters," in *MASCOTS '05: Proceedings of the 13th IEEE International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 23–34.

[11] J. Dean, J. E. Hicks, C. A. Waldspurger, W. E. Weihl, and G. Chrysos, "Profileme: hardware support for instruction-level profiling on out-of-order processors," in *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 1997, pp. 292–302.

[12] S. S. Shende and A. D. Malony, "The tau parallel performance system," *Int. J. High Perform. Comput. Appl.*, vol. 20, no. 2, pp. 287–311, 2006.

[13] Intel, "Vtune," http://www.intel.com/cd/software/products/asmo-na/eng/vtune/239144.htm.

[14] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz, "Performance analysis using the mips r10000 performance counters," in *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*. Washington, DC, USA: IEEE Computer Society, 1996, p. 16.

[15] D. Kim, J. Eom, and C. Park, "L4oprof: a performance-monitoring-unit-based software-profiling framework for the l4 microkernel," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 4, pp. 69–76, 2007.

[16] J. Mellor-Crummey, R. Fowler, and D. Whalley, "Tools for application-oriented performance tuning," in *ICS '01: Proceedings of the 15th international conference on Supercomputing*. New York, NY, USA: ACM, 2001, pp. 154–165.

[17] A. R. Lebeck and D. A. Wood, "Cache profiling and the spec benchmarks: A case study," *Computer*, vol. 27, no. 10, pp. 15–26, 1994.

[18] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod, "Using the simos machine simulator to study complex computer systems," *ACM Trans. Model. Comput. Simul.*, vol. 7, no. 1, pp. 78–103, 1997.

[19] L. D. Rose, Y. Zhang, and D. A. Reed, "Svpablo: A multi-language performance analysis system," in *TOOLS '98: Proceedings of the 10th International Conference on Computer Performance Evaluation: Modelling Techniques and Tools*. London, UK: Springer-Verlag, 1998, pp. 352–355.

[20] L. Noordergraaf and R. Zak, "Smp system interconnect instrumentation for performance analysis," in *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on Supercomputing*. Los Alamitos, CA, USA: IEEE Computer Society Press, 2002, pp. 1–9.

[21] Intel, "Intel fortran and c++ compiler professional editions for linux," http://www.intel.com/cd/software/products/asmo-na/eng/compilers/282048.htm.

[22] GNU, "The gnu compiler collection," http://gcc.gnu.org/.

[23] "Options for code generation," http://gcc.gnu.org/onlinedocs/gcc-4.3.2/gcc/Code-Gen-Options.html#Code-Gen-Options.

[24] SystemG, "Systemg," http://www.top500.org/system/9833.

[25] S. Eranian, "Perfmon kernel patch," http://sourceforge.net/project/showfiles.php?group_id=144822.

[26] NASA, "The nas parallel benchmarks (npb)," http://www.nas.nasa.gov/Resources/Software/npb.html.

[27] LINPACK, "Linpack," http://www.netlib.org/linpack/.

[28] R. C. Whaley and J. J. Dongarra, "Automatically tuned linear algebra software," in *Supercomputing '98, Proceedings of the 1998 IEEE/ACM Conference on Supercomputing*, Nov 1998.

[29] K. Goto and R. A. van de Geijn, "Anatomy of high-performance matrix multiplication," *ACM Transactions on Mathematical Software (TOMS)*, vol. 34, no. 3, May 2008.

[30] K. Goto and R. van de Geijn, "On reducing tlb misses in matrix multiplication," The University of Texas at Austin, Department of Computer Sciences, Tech. Rep. CS-TR-02-55, 2002.